Instituto Superior de Engenharia do Porto
Departamento de Engenharia Informática

SISTEMAS OPERATIVOS I

Textos de Apoio às Aulas Práticas

Ferramentas de Desenvolvimento e Instalação de Aplicações

extraído do livro
"LINUX Curso Completo"
Fernando Pereira

Novembro de 2002

Lino Oliveira

Sugestões e participações de erros para: lino@dei.isep.ipp.pt

FERRAMENTAS DE DESENVOLVIMENTO E INSTALAÇÃO DE APLICAÇÕES

Índice

1 FERRAMENTAS DE DESENVOLVIMENTO

Muito antes de o *Linux* aparecer, já existiam dezenas de ferramentas de desenvolvimento *open source,* que haviam sido criadas pela *Free Software Foundation.*

Por essa razão, as primeiras aplicações que foram transportadas para o *Linux,* foram as ferramentas GNU. Foram estas ferramentas que possibilitaram a adaptação e o desenvolvimento de todos os restantes programas que se seguiram.

Na verdade, quando Linus Torvalds afixou a primeira versão de *Linux* na *Internet,* uma das características mais importantes que foram anunciadas, foi o facto de o sistema correr o compilador de linguagem C «gcc». Depois de este anúncio ter sido feito, a evolução que o *Linux* seguiu é bem conhecida e deve-se muito às suas ferramentas de desenvolvimento.

1.1 LINGUAGENS DE PROGRAMAÇÃO DISPONÍVEIS

Dentro de todas as ferramentas de desenvolvimento, as mais importantes são as linguagens de programação e os respectivos compiladores.

Os programas são habitualmente desenvolvidos usando linguagens de programação de alto nível, que são relativamente fáceis de entender pelas pessoas e em especial pelos programadores.

Pelo contrário, os microprocessadores só conhecem a linguagem máquina, que é composta exclusivamente por sequências de números. Por essa razão, a linguagem máquina é bastante difícil de utilizar pelos programadores.

Os compiladores são programas que traduzem ficheiros de código-fonte, escrito em linguagens de programação de alto nível, para ficheiros binários em linguagem máquina.

Cada linguagem de programação necessita de um compilador específico, que entende as suas regras de sintaxe.

As distribuições de *Linux* costumam incluir os seguintes compiladores:

Compilador	Linguagem	Descrição
cc gcc egcs	C	Compiladores de linguagem C do projecto GNU. O nome tradicional é «cc», mas em alternativa pode ser usado com os nomes «gcc» e «egcs».
g++	C++	Compilador de C++ do projecto GNU. Faz parte do mesmo conjunto que o «gcc».
objc	Objective C	Compilador de C++ do projecto GNU. Faz parte do mesmo conjunto que o «gcc».
F77	Fortran	Compilador de *Fortran* do projecto GNU.
guavac	Java	Compilador de *Java* do projecto GNU.
jikes	Java	Compilador de *Java* oferecido pela IBM.
javac	Java	Compilador de *Java* original, desenvolvido pela Sun Microsystems.
as86	Assembly	Assemblador para os CPUs da família *Intel* 80x86 e *Pentiums.*

cpp	C/C++	O pré-processador de linguagem C. Geralmente só é chamado pelo próprio «gcc» e nunca é usado directamente pelo utilizador.
p2c	Pascal	Tradutor de linguagem *Pascal* para C ou, C++. Converte programas em *Pascal* para C.
f2c	Fortran	Tradutor de linguagem *Fortran* para C. Converte programas em *Fortran* para C.
ld	Linker	Constrói programas executáveis a partir de vários ficheiros binários em formato *objecto.*
Java	Java	Interpretador de Java *bytecodes* original, criado pela SUN. Permite correr programas executáveis em *Java.*
kaffe	Java	Interpretador de *Java bytecodes* alternativo. Permite correr programas executáveis em *Java.*

Para além destas linguagens, também é possível obter compiladores e interpretadores *open source* de *Ada, Basic, Lisp, Cobol, Prolog, Forth, Scheme* e muitas outras linguagens.

Qualquer destes pacotes pode ser facilmente encontrado nos índices de *software* para *Linux* dos *sites* habituais. Dois bons locais para os encontrar, são http://www.gnu.org e http://www.freshmeat.net (na secção de ferramentas de desenvolvimento).

Por outro lado, as linguagens de *scripting Perl, Python* e TCL também são óptimas ferramentas para criar aplicações, que nunca devem ser esquecidas. Apesar de não oferecerem a mesma *performance* que as linguagens compiladas, são mais flexíveis e proporcionam um ciclo de desenvolvimento bastante mais curto, que permite criar aplicações muito rapidamente.

Finalmente, para os projectos mais complexos, existem outras ferramentas como o *Lex* («flex») e *Yacc* («bison»/«gyacc»), que permitem construir compiladores e interpretadores de mais linguagens de programação. O Yacc é conhecido como o compilador de compiladores ...

1.2 EXEMPLO EM LINGUAGEM C

0 seguinte programa em linguagem C apenas escreve uma mensagem de texto no ecrã, a dizer «Hello World!».

Ficheiro: «helloworld.c»:

```
#include <stdio.h>

int main( int argc, char** argv)
{
      printf( "Hello World !\n" );
}
```

A compilação do programa anterior pode ser realizada com o programa «gcc», ou com um dos seus pseudónimos «cc» e «egcs»:

```
$ gcc helloworld.c
```

Por omissão, o programa executável criado pelo «gcc» recebe sempre o nome «a.out».

Desta forma, para correr o programa que acabámos de compilar, é necessário escrever o seu nome na linha de comandos:

```
$ a.out
Hello World
```

O «gcc» tem dezenas de opções, que permitem optimizar o código binário para maximizar a velocidade, activar mensagens de aviso e detecção de erros, seleccionar várias normas e variantes de C, escolher o tipo de CPUs usado, etc. Como sempre, a página de manual do «gcc» inclui toda a documentação necessária.

Exemplo:

```
$ cc -02 -Wall -m486 -o olamundo helloworld.c
```

As opções usadas foram as seguintes:

-O2	Instruir o compilador para gerar código-máquina optimizado para maximizar a velocidade.
-Wall	Activa todas as mensagens de aviso.
-m486	Utilizar o co-processador aritmético incluído em todos os CPUs da família *Intel* x86 a partir do i486.
-o	Definir o nome do ficheiro gerado. O programa criado pelo «gcc» deixa de se chamar «a.out» e passa a chamar-se «olamundo».

1.3 CRIAR PROJECTOS COM O «MAKE»

Quando os projectos de *software* se tornam demasiado complexos, é necessário utilizar ferramentas para os gerir de forma automática.

Depois do compilador de C, o «make» é a ferramenta do *Linux* que os programadores mais utilizam, porque permite automatizar todos os passos da construção e instalação de aplicações em código-fonte.

Para funcionar, o «make» necessita de um ficheiro *(Makefile)* com uma descrição de todos os passos necessários para construir uma aplicação.

Por exemplo, vamos supor que pretendíamos construir uma aplicação chamada «contab», que é composta por 5 ficheiros em código-fonte: «principal.c», «menus.c», «contas.c», «impressao.c» e «def.h».

Para construir a aplicação, é necessário compilar todos os ficheiros fonte «.c» e construir ficheiros objecto «.o». Depois de todos os ficheiros «.o» estarem criados, têm de ser juntos num único ficheiro executável, usando o *linker* do sistema («ld»).

A Makefile usada para construir esta aplicação é a seguinte:

```
# Makefile

contab: principal.o menus.o contas.o impressao.o
      gcc -o contab principa1.o menus.o contas.o impressao.o -lm

principal.o: principal.c def.h
      gcc -c -O2 principal.c
```

```
menus.o: menus.c def.h
	gcc -c -O2 menus.o

contas.o: contas.c def.h
	gcc -c -O2 -m486 -ffast-math contas.c

impressao.o: impressao.c def.h
	gcc -c -O2 impressao.c

clean:
	rm -f *.o
	rm -f core
	rm -f contab

rebuild: clean contab
```

A Makefile descreve uma lista de dependências entre ficheiros, começando nos ficheiros de código-fonte até chegar ao objectivo final, que é o programa executável.

Para melhor compreender o seu funcionamento, observemos a primeira regra de dependência:

```
contab: principal.o menus.o contas.o impressao.o
	gcc -o contab principa1.o menus.o contas.o impressao.o -lm
```

Esta regra diz que o programa principal «contab» depende dos ficheiros objecto «principal.o», «menus.o», «contas.o» e «impressao.o».

Quando o «make» entra em funcionamento, verifica se o ficheiro «contab» existe e se a sua data de alteração é posterior à dos ficheiros de que ele depende.

O programa final «contab» só necessita de ser reconstruído se algum dos ficheiros «.o» tiver sido alterado depois de este ter sido gerado ou, obviamente, se o ficheiro «contab» nem sequer existir.

A seguir à primeira linha de cada regra, seguem-se várias linhas contendo os comandos necessários para construir os ficheiros em questão.

Na regra acima indicada, é utilizado o comando «gcc -o contab principal.o menus.o contas.o impressao.o -lm» para *linkar* (juntar) todos os ficheiros objecto num único programa executável. A opção «-o contab» define o nome do programa resultante e a opção «-lm» serve para adicionar ao programa as funções matemáticas definidas na biblioteca «/lib/libm.a».

As regras seguintes da *Makefile definem* as dependências de cada ficheiro objecto em relação aos ficheiros de código-fonte iniciais. Se algum destes dois ficheiros for alterado pelo utilizador, então o ficheiro *objecto* correspondente tem de ser reconstruído e, por arrasto, o programa final também.

Por exemplo, o ficheiro «menus.o» depende do «menus.c» e do «defs.h».

```
principal.c: principal.c def.h
	gcc -c -O2 principal.c
```

Mais uma vez, o comando usado para construir os ficheiros objecto é o compilador «gcc», mas desta vez foi usada a opção «-c» para evitar que o compilador tente *linkar o* ficheiro resultante. Desta forma, em vez de criar um ficheiro executável, o «gcc» limita-se a criar um ficheiro objecto chamado «principal.o».

No final da *Makefile* existem duas regras adicionais, que servem para fazer manutenção ao projecto.

A regra *clean* não possui quaisquer dependências, pois apenas é usada para automatizar a limpeza do projecto. Quando o utilizador escreve o comando «make clean», o programa «rm» é executado várias vezes para remover todos os ficheiros objecto, o programa executável e eventuais ficheiros de erros *(core).*

A regra *rebuild* funciona da forma exactamente oposta à *clean:* tem dependências, mas não define quaisquer instruções para ser executadas. Quando o utilizador escreve o comando «make rebuild», é aplicada a regra *clean* e em seguida o programa «contab» é construído novamente.

A sintaxe das *Makefiles* é bastante complexa e possui bastantes opções para automatizar muitas tarefas. Por exemplo, as regras da *Makefile anterior* que definem as dependências dos ficheiros objecto em relação aos ficheiros de código-fonte, poderiam ter sido abreviadas numa única regra:

```
.c.o:
      gcc -c -02 $<
```

Esta regra define uma dependência genérica de todos os ficheiros *objecto* em relação aos ficheiros *fonte*, baseada nas extensões dos seus nomes. Desta forma, o «make» fica a saber que um ficheiro com a extensão «.o» deve ser construído a partir de outro ficheiro com o mesmo nome, mas com a extensão «.c».

O comando usado para construir os ficheiros «.o» continua a ser o «gcc», mas desta vez foi usado o símbolo «$<» que representa o nome do ficheiro «.c» usado em cada caso particular.

1.4 O GNU AUTOCONF

O universo UNIX está fragmentado em dezenas de versões do sistema, que em geral são compatíveis umas com as outras, mas cada uma delas possui pequenas especificidades que as tornam diferentes de todas as outras.

Mesmo no *Linux,* existem várias distribuições e diversas versões do *kernel,* que possuem pequenas diferenças entre si. Por exemplo, as directorias usadas para organizar as aplicações, as bibliotecas e os ficheiros partilhados, variam de distribuição para distribuição.

Durante muitos anos, fazer aplicações portáveis que funcionassem em todas as versões destes sistemas operativos era um autêntico pesadelo.

Contudo, desde que apareceu o GNU «autoconf», este problema tornou-se bastante mais fácil de resolver.

O «autoconf» analisa exaustivamente o sistema operativo e detecta todas as suas potencialidades, para além da sua compatibilidade com os componentes e as normas de *software* utilizadas em cada aplicação.

Depois de analisar o sistema, o «autoconf» gera um ficheiro contendo uma lista de definições (config.h) que descrevem todas as capacidades oferecidas pelo sistema, incluindo as variantes de cada norma de *software* que foram detectadas.

Durante a sua compilação, as aplicações podem utilizar as definições contidas no ficheiro «config.h», para gerar código condicional, que se adapta às características de cada versão do sistema operativo.

Desta forma, torna-se bastante fácil criar aplicações portáveis, que podem funcionar em virtualmente todas as versões de UNIX e em outros sistemas operativos.

A maioria das aplicações *open source* utilizam o *Autoconf* e por essa razão conseguem adaptar-se automaticamente a cada distribuição de *Linux* e até podem ser recompiladas noutras versões de UNIX sem grandes problemas.

Antes de compilar uma aplicação baseada no *Autoconf,* é necessário executar um *script* chamado «configure» (ou «autogen.sh»).

O *script* «configure» parte de um ficheiro chamado «configure.in» e gera o ficheiro de definições «config.h», que é usado durante a compilação das aplicações. Para além disso, também pode ser usado para construir automaticamente as *Makefiles,* usando o utilitário «automake».

O «automake» pode ser uma ajuda preciosa, porque uma parte significativa do conteúdo das Makefiles praticamente não varia de aplicação e pode ser gerado de forma automática. Um bom exemplo é o procedimento de instalação, que geralmente se limita a copiar os ficheiros finais para as directorias correctas do sistema.

1.5 TESTE E DEPURAÇÃO DE APLICAÇÕES

O depurador de erros mais usado no *Linux* é o «gdb» (GNU *Debugger).*

O «gdb» tem uma *interface* baseada em comandos de texto, que pode parecer difícil de usar pelas pessoas habituadas aos ambientes de programação visuais, mas depois de conhecer os comandos, torna-se relativamente fácil.

Como acontece com a maior parte dos utilitários oferecidos no *Linux, o* «gdb» é muitíssimo poderoso e possui dezenas de comandos que permitem avançar passo a passo, usar *breakpoints, tracepoints e breakpoints* condicionais, visualizar o valor de variáveis e expressões, modificar variáveis, analisar o *stack,* executar partes isolados de programas, etc.

Para que uma aplicação possa ser depurada, é necessário que tenha sido compilada com uma opção especial («-g»), que adiciona informação suplementar ao programa executável, contendo o nome de todas as variáveis, funções e o número de cada linha correspondente a cada zona do código gerado.

Exemplo:

```
$ gcc -g helloworld.c -o olamundo
$ gdb olamundo
(gdb) break main
(gdb) run
(gdb) help
```

Uma das particularidades mais úteis do «gdb» é o facto de permitir analisar o conteúdo dos ficheiros *core*, que são gerados pelo sistema quando as aplicações rebentam devido a erros.

Desta forma, podemos descobrir o ponto exacto onde as aplicações rebentaram, ver o conteúdo que as variáveis tinham, listar as funções que haviam sido chamadas, etc.

Por exemplo, para descobrir o ponto exacto onde rebentou um programa chamado «prog1», bastaria executar «gdb prog1 core» e usar o comando interno do «gdb»: «where».

Outro programa muito importante é o «gprof», que é muito útil para os programadores optimizarem a performance das suas aplicações.

Para usar o «gprof» é necessário compilar os programas com uma opção especial do compilador: «-p» ou «-pg».

Quando os programas compilados com esta opção são executados, geram um ficheiro «gmon.out» contendo todos os tempos de execução de cada função do programa, sempre que estas são chamadas.

O programa «gprof» é usado para processar o ficheiro de tempos e criar um relatório com as estatísticas das funções que estão a gastar mais tempo e foram chamadas mais vezes.

Com base neste relatório, os programadores podem concentrar os seus esforços a optimizar apenas as partes do programa que estão realmente a ocupar mais tempo, uma vez que as outras são irrelevantes.

A classe de erros que geralmente é mais difícil de corrigir e detectar são os erros de memória dinâmica. Para ajudar a resolver este tipo de erros, existem dois pacotes de *software* chamados *Electric Fence* e *Checker*.

Tanto o *Electric Fence* como o *Checker* permitem descobrir zonas de memória que são reservadas e nunca mais voltam a ser libertadas, zonas de memória que são usadas depois de serem libertadas, zonas de memória que são perdidas, acessos a zonas de memória fora dos limites reservados, etc.

Finalmente, temos os programas «*strace*» e «*ltrace*», que permitem obter a lista de chamadas ao *kernel* do sistema e as bibliotecas dinâmicas usadas pelos programas analisados.

1.6 MANIPULAÇÃO DE BIBLIOTECAS E FICHEIROS EXECUTÁVEIS

As bibliotecas são constituídas por um arquivo contendo uma colecção de ficheiros *objecto («*.o»)* e um índice contendo a lista de símbolos (funções e variáveis) definidos em todos esse ficheiros.

Para construir uma biblioteca, são utilizados dois comandos: o «ar», para criar o arquivo propriamente dito e o «ranlib» para criar o índice de símbolos. 0 comando «ar» é muito parecido ao «tar» que costuma ser usado nos *backups.*

Por exemplo, para criar uma biblioteca chamada «libutil.a» a partir de quatro ficheiros objecto «obj1.o», «obj2.o», obj3.o» e «obj4.o», poderíamos proceder da seguinte forma:

```
$ ar cv libutil.a obj1.o obj2.o obj3.o obj4.o
$ ranlib libutil.a
```

Para além dos comandos anteriores, existem mais alguns utilitários que permitem manipular ficheiros objecto e executáveis:

nm	Mostra a lista de símbolos definidos em ficheiros objecto e em bibliotecas.
ar	Arquivador de ficheiros objecto, para construir bibliotecas.
ranlib	Cria um índice com os símbolos definidos numa biblioteca.
ldconfig	Cria um índice de todas as bibliotecas dinâmicas presentes no sistema, tendo em conta as suas versões.
strip	Retira toda a informação usada para fazer o *debug* dos programas executáveis.
strings	Mostra uma lista com todas as frases de texto contidas num programa executável.
ldd	Mostra a lista de bibliotecas dinâmicas usadas por um programa.
objcopy	Copia dados de um ficheiro objecto para outro.
size	Mostra a dimensão de cada uma das regiões de um ficheiro executável (código, dados, dados inicializados, etc.).

1.7 CONTROLO DE VERSÕES

Nos projectos de *software* de grande dimensão, o controlo de versões pode tornar-se um problema relativamente complicado, pois é necessário guardar cópias de todas as versões antigas e conjugar as alterações produzidas por todos os colaboradores.

Uma situação muito frequente, acontece quando existem várias versões do mesmo programa a ser usadas simultaneamente por utilizadores diferentes. Quando um utilizador de uma versão antiga comunica um erro, pode ser necessário recuperar essa versão do programa, para detectar as causas do problema. Contudo, os erros não podem ser corrigidos apenas na versão onde foram detectados, porque muitas vezes também afectam as versões actuais do programa.

Por outro lado, em equipas de desenvolvimento distribuídas, podem haver muitos programadores localizados em zonas afastadas, que nem sempre podem comunicar directamente uns com os outros. Nessas situações, é necessário conjugar as modificações feitas por todos os colaboradores, para evitar conflitos.

Nos dois casos, a solução passa pelo uso de um dos sistemas de controlo de versões que existem no *Linux:* o <<rcs>> *(Revision Control System)* e o <<cvs>> *(Concurrent Versions System).*

Ambos os produtos guardam um histórico das alterações produzidas a cada ficheiro de um projecto e permitem recuperar qualquer versão do mesmo. Quando as equipas de trabalho são compostas por várias pessoas, também é possível gerir conflitos no acesso a ficheiros, juntar as alterações produzidas por pessoas diferentes e, até mesmo, juntar versões diferentes.

Uma terceira forma de fazer a gestão de versões, consiste em utilizar os programas «diff» e «patch».

O «diff» identifica as diferenças entre várias versões do mesmo ficheiro e o «patch» reconstrói uma das versões a partir da outra e das respectivas diferenças.

Apesar de o «diff» e de o «patch» serem muito utilizados para distribuir *software* em código-fonte pela *Internet,* o controlo de versões tem de ser feito manualmente. Pelo contrário, o «rcs» e o «cvs» automatizam todo este processo e por essa razão são substancialmente melhores...

Para obter informação detalhada sobre o «rcs» e o «cvs», e aconselhável ler as páginas de manual «rcsintro» e «cvs».

1.8 OUTROS UTILITÁRIOS

Para além dos programas anteriores, existem mais alguns utilitários, que são utilizados muito frequentemente pelos programadores:

grep	Procura palavras e expressões regulares numa lista de ficheiros.
diff	Gera um ficheiro com as diferenças entre vários ficheiros.
patch	Constrói um ficheiro a partir de outros e das respectivas diferenças, produzidas pelo «diff».
indent	Indenta automaticamente um ficheiro contendo código-fonte em linguagem C/C++.
cproto	Ajuda a converter programas escritos em linguagem C tradicional para C ANSI.
ctags/etags	Gera uma lista com todos os símbolos definidos num grupo de ficheiros em código-fonte, que inclui as constantes, variáveis e funções. Mais tarde, a lista de *tags* pode ser usada pelos editores «vi» e «emacs» para encontrar rapidamente o local onde os símbolos foram definidos.

1.9 BIBLIOTECAS

As bibliotecas são um dos recursos mais valiosos dos programadores, porque permitem utilizar os recursos do sistema operativo e utilizar pacotes de *software* desenvolvidos por outros programadores.

As distribuições de *Linux* costumam ser fornecidas com centenas de bibliotecas, que estão localizadas nas directorias «/lib», «/usr/lib», «/usr/local/lib», «/usr/iX86*-linux/lib», «/usr/X11R6/lib» e «/opt/kde/lib».

Existem bibliotecas com funções matemáticas, com chamadas ao sistema operativo, com funções para aceder à rede, com funções para manipular imagem e som, com objectos para construir a *interface* gráfica das aplicações, etc.

O compilador de C e o *linker* do sistema «ld», possuem as opções «-l» e «-L», que permitem utilizar bibliotecas nos programas por eles gerados.

Exemplo:

```
$ cc jogo.c –L/usr/X11R6/lib -lXt -lX11 -lm
```

Neste exemplo, foi compilado o programa «jogo.c» que utiliza três bibliotecas: «Xt», «X11 » e «m». Estes nomes são abreviaturas, que identificam os ficheiros, onde as bibliotecas estão guardadas: «libXt.so», «libX11.so» e «libm.so».

As primeiras duas bibliotecas «Xt» e «X11» fazem parte do sistema de janelas X e por essa razão estão guardadas na directoria «/usr/X11R6/lib». A opção «-L» indica que o *compilador/linker deve* procurar as bibliotecas na directoria «/usr/X11R6/lib», para além das directorias normais do sistema.

Existem dois tipos de bibliotecas: estáticas e dinâmicas. As bibliotecas estáticas possuem o sufixo «.a» e as bibliotecas dinâmicas costumam possuir os sufixos «.sa» *(shared archive) ou «.so» (shared object).*

Quando um programa é compilado e *linkado* com uma biblioteca estática, o *linker* copia todas as funções e variáveis da biblioteca para dentro do ficheiro executável do programa gerado. Como resultado, são gerados ficheiros enormes, que contêm cópias integrais de todas as bibliotecas usadas.

Pelo contrário, quando um programa é *linkado* com uma biblioteca dinâmica, esta não é acrescentada ao ficheiro. Em vez disso, o programa encarrega-se de carregar a biblioteca dinâmica na altura em que for executado.

As bibliotecas dinâmicas têm as seguintes vantagens:

- Dão origem a ficheiros executáveis muito mais pequenos.
- É possível modificar as versões das bibliotecas sem modificar os programas, pois basta mudar a versão da biblioteca que está instalada no sistema.
- As bibliotecas dinâmicas podem ser usadas por vários programas em simultâneo. Desta forma, a biblioteca só é carregada em memória da primeira vez que um programa a utiliza e depois essa zona de memória é partilhada pelos restantes programas que a utilizam.
- Os programas podem mandar carregar e descarregar bibliotecas dinâmicas durante o seu funcionamento.

2 INSTALAÇÃO DE APLICAÇÕES

As aplicações podem ser instaladas de diversas formas, de acordo com o modo como os autores criaram os pacotes de *software.*

As aplicações comerciais costumam ser fornecidas sob a forma de pacotes binários de *software* e as aplicações *open source* costumam estar disponíveis nos dois formatos: binário e código-fonte.

Em geral, os pacotes em formato binário são mais fáceis de instalar do que o código-fonte, porque não é necessário recompilar os programas. Contudo, quando se pretende corrigir erros ou fazer alterações aos programas para os adaptar a necessidades específicas, é obrigatório usar o código-fonte.

2.1 INSTALAÇÃO DE PACOTES BINÁRIOS DE SOFTWARE

Quase todas as distribuições de *Linux* possuem sistemas de gestão dos pacotes de *software* instalados.

A maioria das distribuições, como por exemplo a *Red Hat,* a *Suse* e a *Mandrake* usam o sistema RPM *(Redhat Package Management).* Por outro lado, as distribuições *Debian* e *Corel* utilizam o sistema DEB. Existem ainda os formatos KISS, SLACK e BSD, mas o RPM é o formato mais divulgado.

Ambos os sistemas são extremamente poderosos e versáteis, pois permitem:

- Visualizar a hierarquia de pacotes instalados, organizada por assuntos e obter informação acerca do seu conteúdo.
- Instalar, actualizar e remover pacotes de *software.*
- Verificar as dependências e resolver conflitos entre os pacotes de *software* instalados.
- Testar a integridade dos ficheiros, para detectar ficheiros apagados acidentalmente, modificados por vírus ou substituídos propositadamente por atacantes remotos, com o objectivo de instalar cavalos-de-tróia.

Existem vários programas para gerir pacotes RPM, começando pelo próprio comando «rpm» até aos programas que funcionam no ambiente de janelas X, «glint», «gnorpm» e «kpackage».

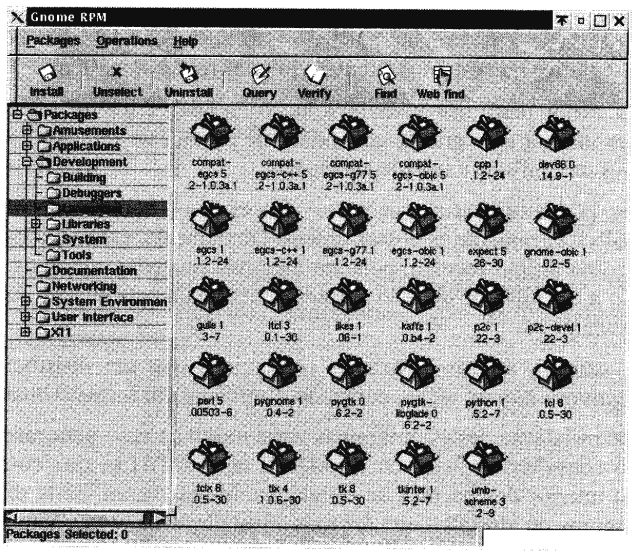

Fig. 1.19 – O programa «gnorpm» é uma interface *gráfica para instalar pacotes RPM.*

Os pacotes DEB, que costumam ser utilizados nas distribuições *Debian* e *Corel,* são manipulados através do programa «dpkg».

O «kpackage» é extremamente versátil e também permite manipular pacotes DEB, KISS, SLACK e BSD.

A forma mais simples de instalar pacotes de *software* consiste em usar os programas que funcionam em X e proceder à instalação utilizado simples cliques de rato. Contudo, os utilizadores avançados, costumam preferir usar o «rpm» directamente na linha de comandos.

Para instalar uma aplicação contida no ficheiro «pacote.rpm»:

```
$ rpm -ih pacote.rpm
```

Para actualizar uma nova versão de uma aplicação:

```
$ rpm -Uvh pacote.rpm
```

Para desinstalar uma aplicação:

```
$ rpm -e pacote
```

Muitas vezes, o «rpm» detecta problemas de dependências entre pacotes de software e recusa-se a prosseguir. Quando isso acontece, o «rpm» costuma ter razão e isso significa que estamos a tentar violar as regras do sistema.

Contudo, na maior parte das vezes a solução “limpa” é muito trabalhosa, pois obrigaria a re-instalar uma grande quantidade de pacotes de software, por forma a resolver todos os conflitos e interdependências.

Nestes casos, podemos usar as opções «--force» e «--nodeps» do «rpm», para que este obedeça cegamente às nossas ordens. Obviamente, estas opções só devem ser usados como último recurso, pois em algumas situações corremos o risco de comprometer a estabilidade de todo o sistema, especialmente quando substituímos bibliotecas dinâmicas.

Finalmente, existem muitos programas comerciais, que são distribuídos sob a forma de arquivos «tar» ou «tgz», contendo o *software* em formato binário.

O método de instalação destes programas varia muito de caso para caso, pelo que é necessário consultar os ficheiros README ou INSTALL, que costumam ser incluídos no interior dos pacotes de instalação. Muitas vezes, basta executar um programa «setup» ou «install», que faz toda a instalação de forma automática.

2.2 INSTALAÇÃO DE APLICAÇÕES A PARTIR DO CÓDIGO-FONTE

Como o seu nome indica, os programas *open source* estão sempre disponíveis em código-fonte.

Para instalar programas a partir do código-fonte, é necessário obter o respectivo arquivo «tar» e descomprimir o seu conteúdo para uma directoria local:

```
$ mkdir tmp-install
$ cd tmp-intall
$ tar xvfz ../pacote.tar.gz
$ ls
$ cd directoria-do-programa
$ ls
```

Depois de descomprimir o *software,* é necessário ler os ficheiros README ou INSTALL, Para conhecer a sequência de comandos necessários para compilar o código-fonte e instalar a aplicação no sistema.

A maior parte dos programas recentes possui um método de instalação baseado no GNU «autoconf» e, por esse motivo, a sequência de comandos de instalação é praticamente sempre igual.

Por essa razão, a seguinte sequência de comandos de instalação já é considerada *um standard* nas aplicações *open source:*

```
$ ./configure
$ make
$ su
# make install
```

Depois de a instalação estar concluída, é possível usar o comando «make clean» para remover todos os ficheiros temporários, que foram usados durante a compilação do código-fonte.

3 NOTA FINAL

Este texto foi extraído do livro "LINUX Curso Completo" de Fernando Pereira, da FCA - Editora de Informática, Lda., correspondendo às secções 1.17 e 1.18.

É um livro cuja compra recomendo vivamente pois cobre de uma maneira exaustiva (sem ser excessivamente técnica) todos os aspectos do *Linux*, sistema operativo para servidores (e não só) com maior crescimento nos últimos anos.

Para além destes aspectos, vem acompanhado com 2 CD's que contêm o *Linux Red Hat* 7.1 completo e mais de 800 pacotes de *software* e muita documentação.